AFRL-IF-RS-TR-2006-173
Final Technical Report
May 2006

PERVASIVE SELF-REGENERATION THROUGH CONCURRENT MODEL-BASED EXECUTION

Massachusetts Institute of Technology, CSAIL

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. S476

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2006-173 has been reviewed and is approved for publication.

APPROVED: /s/

ALAN J. AKINS
Project Engineer

FOR THE DIRECTOR: /s/

WARREN H. DEBANY, Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	MAY 2006	Final Jun 04 – Dec 05

4. TITLE AND SUBTITLE
PERVASIVE SELF-REGENERATION THROUGH CONCURRENT MODEL-BASED EXECUTION

5. FUNDING NUMBERS
C - FA8750-04-2-0243
PE - 62301E
PR - S476
TA - SR
WU - SP

6. AUTHOR(S)
Paul Robertson, Brian C. Williams

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Massachusetts Institute of Technology, CSAIL
32 Vassar Street
Cambridge Massachusetts 02139

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency AFRL/IFGA
3701 North Fairfax Drive 525 Brooks Road
Arlington Virginia 22203-1714 Rome New York 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-2006-173

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Alan J. Akins/IFGA/Alan.Akins@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

The model-based approach to cognitive immunity of computer software involves, at multiple levels, the use of models to describe correct behavior of the system at appropriate level of abstraction. By specifying the correct operation of the system at levels of the abstract states, this allows the system to select between redundant methods of achieving those states and thereby allowing the system to be robust to perturbations inherent in the environment in which the system operates. Our system, which was demonstrated on a robotic platform, was able to successfully execute a complex robotic mission even in the face of software component failure and unexpected perturbations in the physical environment by (a) reconfigure its software components, and (b) selecting different redundant methods when faced with failing software components.

14. SUBJECT TERMS
Regenerative Systems, Cyber Defense, Software Reconfiguration, Cognitive Immunity, Model-based Execution

15. NUMBER OF PAGES
37

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures

1. Introduction

Getting robots to perform even simple tasks can be fraught with difficulty. Sending a robot to a distant planet, such as Mars, raises a number of significant issues concerning autonomy. The intended mission itself may be quite complex, conceivably involving coordinated work with other robots. An example of such an anticipated mission is to send a team of robots to Mars where they will, among other things, construct habitat that will be used by astronauts that arrive at a later time. Quite apart from the complexity of the mission, these robots are likely to be expensive especially when considering the cost of transporting them to the planet surface. The loss of a robot would therefore be a serious loss that could jeopardize the feasibility of the astronaut arrival. For this reason robotic explorations (Figure 1) of Mars (Sojourner, Spirit, and Opportunity) have not been autonomous except for very minor navigation capability.

Figure 1 - NASA Rover Spirit

Operating rovers from earth is an excruciating process because the time delay caused by the time it takes a signal to travel between Earth and Mars does not permit tele-operation of the rovers. Consequently a large team of mission planners must plan and test, in excruciating detail, plans for the next day's operations. This approach will not work for more complex missions such as cooperative robots assembling an astronaut habitat.

In our rover testbed we are attempting to perform such complex cooperative missions using a team of iRobot ATRV robots (Figure 2a) with arms mounted on them for instrument placement/collection and construction tasks.

The hostile nature of space and the uncertainty of the Martian environment mean that we must consider repair of a mission plan as well as adapting to failed components. In modern systems these failed systems are as likely to be software systems as hardware. Indeed many of NASA's recent problems have been software problems. In this paper we describe our approach to dealing with perturbations with a particular emphasis to the software/mission planning aspect of robot autonomy.

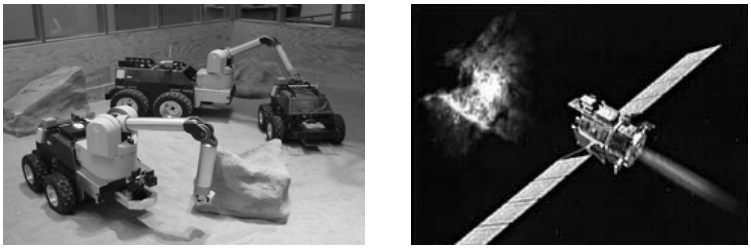


Figure 2 - (a) MIT MERS Rover Testbed, (b) Deep Space 1: Fight Experiment

In complex, concurrent critical systems, every component is a potential point of failure. Typical attempts to make such systems more robust and secure are both brittle and incomplete. That is, the security is easily broken, and there are many possible failure modes that are not handled. Techniques that expand to handling component level failures are very expensive to apply, yet are still quite brittle and incomplete. This is not because engineers are lazy – the sheer size and complexity of modern information systems overwhelms the attempts of engineers, and myriad methodologies, to systematically investigate, identify, and specify a response to all possible failures of a system.

Adding dynamic intelligent fault awareness and recovery to running systems enables the identification of unanticipated failures and the construction of novel workarounds to these failures. Our approach is pervasive and incremental. It is pervasive in that it applies to all components of a large, complex system – not just the "firewall" services. It is incremental in that it coexists with existing faulty, unsafe systems, and it is possible to incrementally increase the safety and reliability of large systems. The approach aims to minimize the cost, in terms of hand-coded specifications with respect to how to isolate and recover from failures.

There are many reasons why software fails. Among the more common reasons are the following:

1. Assumptions made by the software turn out not to be true at some point. For example, if a piece of software must open a file with a given path name it will usually succeed but if the particular disk that corresponds to the path name fails the file will not be accessible. If the program assumed that the file was accessible the program will fail. In highly constrained situations it is possible to enumerate all such failures and hand code specific exception handlers – and such is the standard practice in the industry. In many cases however, particularly in embedded applications, the number of ways that the environment can change becomes so large that the programmer cannot realistically anticipate every possible failure.

2. Software is attacked by a hostile agent. This form of failure is similar to the first one except that change in the environment is done explicitly with the intent to cause the software to fail.

3. Software changes introduce incompatibilities. Most software evolves over its lifetime. When incompatibilities are inadvertently introduced software that previously did not fail for a given situation may now fail.

Whatever the reason for the software failure, we would like the software to be able to recognize that it has failed and to recover from the failure. There are three steps to doing this: Noticing that the software has failed; Diagnosing exactly what software component has failed; and Finding and alternative way of achieving the intended behavior.

In order for the runtime system to reason about its own behavior and intended behavior in this way certain extra information and algorithms must be present at runtime. In our system these extra pieces include models of the causal relationships between the software components, models of intended behavior, and models of correct (nominal) execution of the software. Additionally

models of known failure modes can be very helpful but are not required. Finally the system needs to be able to sense, at least partially, its state, it needs to be able to reason about the difference between the expected state and the observed state and it need to be able to modify the running software such as by choosing alternative runtime methods.

Building software systems in this way comes with a certain cost. Models of the software components and their causal relationships that might otherwise have existed only in the programmers head must by made explicit, the reasoning engine must be linked in to the running program, and the computational cost of the monitoring, diagnosis, and recovery must be considered. In some systems the memory footprint and processor speed prohibit this approach. More and more however memory is becoming cheap enough for memory footprint to not be an issue and processor power is similarly becoming less restrictive. While the modeling effort is an extra cost there are benefits to doing the modeling that offset its cost. Making the modeling effort explicit can often cause faults to be found earlier than would otherwise be the case. The developers can choose the fidelity of the models. More detailed models take more time to develop but allow for greater fault identification, diagnosis, and recovery. Finally our approach to recovery assumes that there is more than one way of achieving a task. The developer therefore must provide a variety of ways of achieving the intended behavior.

The added costs of building robust software in this way are small when compared to the benefits. Among the benefits it allows us to:

1. Build software systems that can operate autonomously to achieve goals in complex and changing environments;
2. Build software that detects and works around “bugs” resulting from incompatible software changes;
3. Build software that detects and recovers from software attacks; and Build software that automatically improves as better software components and models are added.

1.1 Overview of the Approach

Our solution is the ability to add dynamic intelligent fault awareness and recovery to running systems. This dynamic fault awareness is able to identify unanticipated failures and construct novel workarounds to these failures. Furthermore, our approach is *pervasive* and *incremental.* Our approach to software robustness is pervasive in that it applies to *all* components of a large, complex system – not just the “firewall” services. Our approach is incremental in that it coexists with existing faulty, unsafe systems, and it is possible to incrementally increase the safety and reliability of large systems. Finally, our approach can be applied at minimum cost, in terms of hand coded specifications with respect to how to isolate and recover from failures.

2. Technical Approach

At the heart of our system is a model-based programming language called RMPL that provides a language for specifying correct and faulty behavior of the systems software components. The novel ideas in our approach include **method deprecation** and **method regeneration** in tandem with an intelligent runtime model-based executive that performs **automated fault management** from engineering models, and that utilizes decision-theoretic method dispatch. Once a system has been enhanced by abstract models of the nominal and faulty behavior of its components, the model-based executive monitors the state of the individual components according to the models. If faults in a system render some methods (procedures for accomplishing individual goals) inapplicable, method deprecation removes the methods from consideration by the decision-theoretic dispatch. Method regeneration involves repairing or reconfiguring the underlying services that are causing some method to be inapplicable. This regeneration is achieved by reasoning about the consequences of actions using the component models, and by exploiting functional redundancies in the specified methods. In addition, decision-theoretic dispatch continually monitors method performance and dynamically selects the applicable method that accomplishes the intended goals with maximum safety, timeliness, and accuracy.

Beyond simply modeling existing software and hardware components, we allow the specification of high-level methods. A method defines the intended state evolution of a system in terms of goals and fundamental control constructs, such as iteration, parallelism, and conditionals. Over time, the more that a system's behavior is specified in terms of model-based methods, the more that the system will be able to take full advantage of the benefits of model-based programming and the runtime model-based executive. Implementing functionality in terms of methods enables method prognosis, which involves proactive method deprecation and regeneration, by looking ahead in time through a temporal plan for future method invocations.

Our approach has the benefit that every additional modeling task performed on an existing system makes the system more robust, resulting in substantial improvements over time. As many faults and intrusions have negative impact on system performance, our approach also improves the performance of systems under stress.

Our approach provides a well-grounded technology for incrementally increasing the robustness of complex, concurrent, critical applications. When applied pervasively, model-based execution will dramatically increase the security and reliability of these systems, as well as improve overall performance, especially when the system is under stress.

Creating Pervasive Concurrent Fault Aware Processes

This project enables pervasive, concurrent, self regenerative systems that are robust to faults or other unexpected events, and that can be affordably created. Traditional approaches to achieving robustness in complex systems attempt to create closed systems in which the internal elements of the system are carefully engineered over decades to be robust, and where technology for

defensive or fault adaptive behavior is focused on the perimeter. In open systems, however, failures can occur within *any* subsystem, process or component of the system, not just at its perimeter.

To achieve robustness for open systems, we enable *every* process to be *fault aware*, by recognizing and adapting to failure. In contrast to traditional, centralized approaches, our approach supports the creation of *fault-aware processes*. Some of these fault aware processes operate concurrently while communicating across a network, while others operate through a layered architecture within a parent fault aware process.

A valuable case study in creating self regenerative systems is the development of deep space probes, such as the Cassini probe that arrives at Saturn in 2004. These one of a kind space probes frequently experience failures in a wide range of components, due to the novelty of the hardware and space environment, and the length and complexity of the mission. Hence, in contrast to closed systems with a perimeter defense, space systems must be prepared to respond to a failure in *any of its internal components*.

To increase the chance of mission success, mission teams employ two methodologies for allowing a mission to regenerate its intended function. First, flight software teams have developed onboard *fault management systems.* These onboard systems are designed to detect, diagnose and recover from any mission critical, single point of failure, while using at least two sensors to corroborate any diagnosis. They accomplish this objective by detecting misdiagnoses due to corrupted sensors, by invoking component repairs, such as resets, and by reconfiguring hardware in order to isolate and navigate around failures. These diagnoses and recoveries are traditionally encoded as a set of If-Then rules.

Second, to recover from failures when a direct reconfiguration or repair is unavailable, mission operators design extensive sets of contingency procedures that provide several functionally different ways of achieving each mission segment. Suppose that the operators detect a failure event, and that they determine that it will prevent the successful execution of a procedure that is planned to be executed in the future. The operators then replace it with a contingency procedure, which is selected based on two criteria: first, it maximizes science return based on the spacecraft's current performance assessment, and second, the operators can validate that the procedure will execute correctly, given their current knowledge of the health of the spacecraft.

The objective of our project was to enable a vast number of large scale, complex systems to achieve a level of robustness similar to the Cassini space probe. This has been accomplished by building these complex systems out of a set of *fault aware processes,* each implementing a robust service. Each fault aware process coordinates a set of underlying component services by employing a *model-based executive,* which implements the two regeneration methodologies described above, but in a fully automated manner.

More specifically, our model-based executive performs system assessment through fault monitoring, diagnosis and prognosis, using a capability called *mode estimation*. Given this assessment the model-based executive then attempts to repair or reconfigure its underlying component services in order to regenerate any lost function. This function is performed by the

executive's *mode reconfiguration capability.* In light of the new state health determined by mode estimation, the *control sequencer* of the model-based executive automatically analyzes the methods it plans to execute, determine any method whose execution will fail or be suboptimal, and then select a new set of methods that it proves will achieve correctness and optimality. We refer to this as *safe, decision-theoretic dispatch*.

Achieving this level of robustness on a wide scale is limited by the cost of incorporating fault adaptivity. Achieving the level of robustness of missions like Cassini using traditional programming methods has proven to be inordinately expensive. Cassini took seven years to develop, involved hundreds of programmers and mission operators, and cost roughly one billion dollars. NASA efforts to reduce this price to a quarter of a billion dollars have led to catastrophic failures, such as Mars Climate Orbiter and Polar Lander.

To allow fault aware processes to be created at minimal additional cost, we have developed a paradigm called *model-based programming*. A model-based program is similar to a traditional concurrent, embedded program, but is elevated above the level of specifying detailed monitoring, diagnosis and repair procedures. Instead, our model-based executive automatically synthesizes these regeneration tactics by reasoning from models of the program's underlying component services.

Using model-based programming, large-scale self-regenerative systems can evolve overtime through incremental adoption, allowing cost to be amortized. Our previous model-based executive, called Livingstone, was layered on top the NASA Deep Space One flight software, without modification to the underlying flight software. During an onboard flight experiment in May of 1999, Livingstone demonstrated the ability to diagnose and repair over half a dozen injected failures within the NASA Deep Space One Probe, allowing it to achieve its mission without interruption [3]. In addition, as part of the DARPA MOBIES program, we demonstrated model-based programming of an executive, by employing the RMPL programming language, and a far more capable model-based executive, called *Titan*, for the creation of a fault aware process for controlling an automotive cooperative cruise control system.

In this project we have extended the above work by enabling the creation of self-regenerative systems with pervasive fault aware processes. We allow fault aware processes to operate concurrently, as well as individually, and allow them to be composed into more complex fault aware processes, through a hierarchy of model-based executives. Each of these executives provides new self-regeneration mechanisms that safely dispatch functionally redundant methods in order to restore and optimize from loss or degradation of component services.

2.1 A Motivating Example

Here we present a simple representative example application and demonstrate how we apply model-based programming techniques to substantially improve the robustness of the example system.

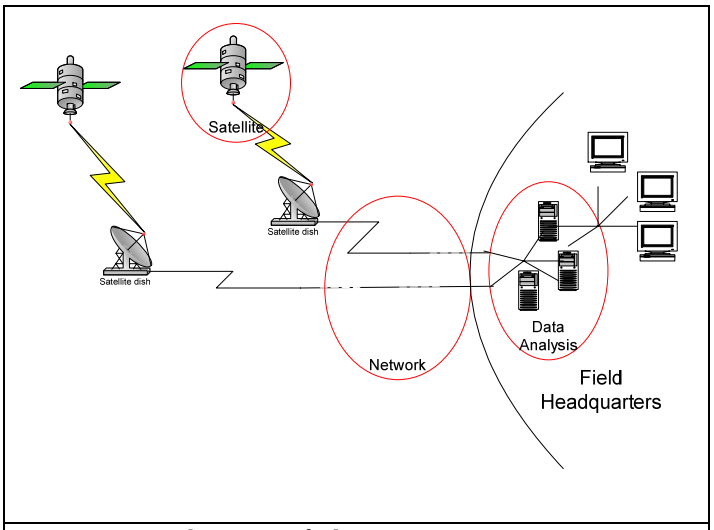


Figure 3 - Architecture of Theater Decision Support System

Example Application: Theater Decision Support System

Consider a complex decision support application that offers a reconnaissance service by assembling data from a wide array of sensors and information flows, analyzing the data, and condensing and presenting the data in real time to officers in charge of theater operations. It is imperative that the data be both **accurate** and **timely.**

The reconnaissance service is built from a network of simpler component services, depicted in Figure 3. Some of the incoming data is gathered from one or more satellites, downlinked to one of a set of ground stations (depending on which satellite and where that satellite is located), transmitted over a network to the field headquarters, then processed by a sequence of image processing algorithms, and finally presented to the officers. There is a myriad of vulnerabilities at every point in Figure 3. We will consider only a few points in this scenario where errors can be injected – either maliciously or by software malfunction (bugs) or hardware faults. At each such point, we describe how services built from fault aware processes can detect and recover from such faults.

Simple Model of Satellite Constellation

Image data is acquired by a set of orbiting satellites, called a constellation. A constellation is a compound service used to support the reconnaissance service. Figure presents a high-level component model of a constellation. Our constellation has two satellites, `sat1` and `sat2`, and each satellite has an associated network connection labeled network. The Constellation component contains a control method get_imagery(region) that transitions a Constellation object into the Image(region) state. Figure gives two implementations of the get_imagery method: one requires that `sat1` is in the correct position, and the second requires that `sat2` be in the correct position. The two method implementations are functionally redundant, in that they both accomplish data acquisition, but by different means.

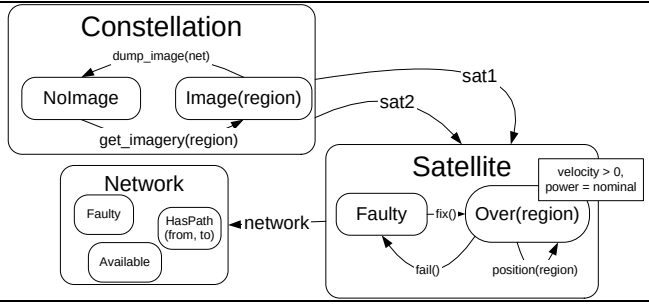


Figure 4 - Simple model of a constellation

```
Constellation::get_imagery(region)
      (sat1 = Over(region)) =>
      Image(region) {
  data = sat1.get_image();
  sat1.download_data(data);
}
Constellation::get_imagery(region)
      (sat2 = Over(region)) =>
      Image(region) {
  data = sat2.get_image();
  sat2.download_data(data);
}
```

Figure 5 - Redundant method definitions for image acquisition

```
1 DecisionSupport::
       get_and_show_imagery(region) {
2    constellation = Image(region);
3    image =
  constellation.network.extract_data();
4    image = Processed;
5    show(image);
6 }
```

Figure 6 - High level method for acquiring imagery from a constellation

```
Satellite::download_data(data, to)
  (sat.network = Available) &
  (sat.network.HasPath(sat.base, to)) { … }
```

Figure 7 - Template of download_data method for Satellites

Figure 6 presents a method in the decision support system that utilizes the constellation to gather image data. Line 2 establishes the goal that the constellation has acquired image data. In our simple model, that goal is achievable directly by invoking the `get_imagery` method, which initiates method dispatch. If both satellites are in position, and therefore both methods are applicable, then method dispatch selects the implementation of `get_imagery` that will return the highest quality data in the timeliest manner. If only one of the satellites is in position, then method dispatch simply invokes the single applicable method.

What if neither satellite is in position? In that case, *Method Regeneration* is charged with finding a plan to satisfy the goal of moving the constellation to Image(region). Since the only path to Image(region) is via the get_imagery method, method regeneration considers the methods implementing get_imagery. Both methods have *preconditions* and *service dependencies*.

In this case, both method implementations have a precondition that their corresponding satellite is in position, and they have service dependencies on the satellite's get_image and

download_data methods. The download_data method (outlined in Figure 7), in turn, has dependencies on the network being available and finding an achievable route. For a dependency on a method to be satisfied, *method prognosis* must determine that there is at least one applicable method at the call site responsible for the dependency. If the networks for both satellites are considered available and routable, then method regeneration invokes `position` on the preferred satellite in order to activate the get_imagery method.

Suppose that both satellites are in position to acquire the desired imagery, but that the network for sat1 is not available. In this case, method dispatch selects the one available method (get_imagery on sat2), but method regeneration applies failure diagnosis and repair to the network of sat1, in order that get_imagery on sat1 becomes available again in the future.

2.2 Fault Aware Processes Through Model-based Programming

To achieve robustness pervasively, fault adaptive processes must be created with minimal programming overhead. *Model-based programming* elevates this task to the specification of the intended state evolutions of each process. A *model-based executive* automatically synthesizes fault adaptive processes for achieving these state evolutions, by reasoning from models of correct and faulty behavior of supporting components.

Each model-based program implements a system that provides some service, such as secure data transmission. This is used as a component within a larger system. The model-based program in turn builds upon a set of services, such as name space servers and data repositories, implemented through a set of concurrently operating components, comprised of software and hardware.

A model-based program has two parts. The first is a *control program,* which uses standard programming constructs to codify specifications of desired system behavior. In addition, to execute the control program, the execution kernel models the component services that it controls. Hence, the second component is a *service model*, which includes models of the component services' nominal behavior and common failure modes. This modeling formalism, called *probabilistic concurrent constraint automata,* unifies constraints, concurrency and Markov processes[3][23].

The control program is a compact specification of intended state evolution that is executed by provably correct synthesis procedures, using knowledge of failure provided by a compact, reusable plant model. A control program is built from a set of methods, specified within an object oriented paradigm. State evolutions are specified by querying and assigning the values of state variables.

Unlike traditional programming, these state variables typically describe states that are hidden within the component services, rather than being directly controllable or observable. Hence an assignment to a state variable corresponds to a goal of achieving that assignment, and a query to a variable corresponds to a request to estimate the variable's state. Assignments are achieved and variables estimated using the *deductive controller* of the Titan model-based executive.

In addition, control program methods are different from traditional programming, in that methods are defined that are functionally redundant, such as the get_imagery methods in Figure 5. These methods provide alternative ways of achieving the same function, and typically rely upon different component services, with different performance profiles. The control sequencer selects from these redundant methods in order to regenerate function when a component service fails, or to improve system performance when a component service degrades. Goal directed method selection endeavors to find an applicable method that will satisfy the current goal, such as image acquisition. Furthermore, *method regeneration* attempts to repair (return to a nominal state) any services that are relied on by methods that can serve active or upcoming goals. For example, our scenario included fault adaptation when each of the satellites became unavailable.

2.3 Component Services Model

The *service model* represents the normal behavior and the known and unknown aberrant behaviors of the program's component services. It is used by a deductive controller to map sensed variables to queried states in the control program and asserted states to specific control sequences. The service model is specified as a concurrent transition system, composed of probabilistic concurrent constraint automata [3]. Each component automaton is represented by a set of component modes, a set of constraints defining the behavior within each mode, and a set of probabilistic transitions between modes. Constraints are used to represent co-temporal interactions between state variables and intercommunication between components. Constraints on continuous variables operate on qualitative abstractions of the variables, comprised of the variable's sign (positive, negative, zero) and deviation from nominal value (high, nominal, low). Probabilistic transitions are used to model the stochastic behavior of components, such as failure and intermittency. Reward is used to assess the costs and benefits associated with particular component modes. The component automata operate concurrently and synchronously.

For example, we model the satellite constellation abstractly as a four-component system (two satellites, each with a network) by supplying the models depicted graphically in Figure 4. Nominally, a satellite can be in one of two modes: Faulty, or Over(region), where region is a continuously varying parameter. The behavior within each of these modes is described by a set of constraints on component variables, such as that the velocity is positive, or that power is nominal. We entertain the possibility of a novel satellite failure by specifying no constraints for the satellite's behavior in the Faulty mode.

A wide range of service modeling formalisms is possible, depending on the category of component services being controlled. For example, Titan uses a discrete probabilistic model to execute the above example. A hybrid model that combines Hidden Markov Models (HMM) and continuous dynamics can be used in order to detect and handle incipient failures [24][25].

2.4 Continuous Fault Adaptation through Model-based Execution

A *model-based executive* automatically synthesizes fault adaptive processes for achieving the state evolutions specified in the control program by reasoning from the models that specify correct and faulty behavior of service components. This synthesis includes methods for transitioning to intended states, monitoring progress, diagnosing failure, and repairing or reconfiguring underlying components. Furthermore, the model-based executive can construct *novel recovery actions* in the face of *novel faults.*

A model-based program is executed by automatically generating a control sequence that moves the component services to the states specified by the control program. We call these specified states *configuration goals*. Program execution is performed using the Titan *model-based executive,* which generates configuration goals and then generates a sequence of control actions that achieve each goal, based on knowledge of the current component service states and model.

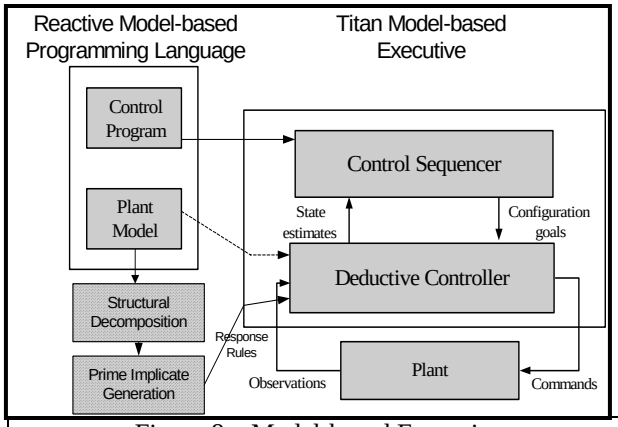


Figure 8 – Model-based Execution

Titan consists of two components, a *control sequencer* and a *deductive controller*. The control sequencer is responsible for generating a sequence of configuration goals, using the control program and plant state estimates. Each configuration goal specifies an abstract state for the component services to achieve. The deductive controller is responsible for estimating the most likely current state of the component services, based on observations from these services (*mode estimation*) and for issuing commands to move the services through a sequence of states that achieve the goals (*mode reconfiguration*).

Control Sequencer

Titan's control sequencer coordinates the component services. When Titan executes the `get_and_show_imagery` method, a goal "constellation = Image(region)" is established. Reconfiguration determines that the only possible way to achieve that goal is via the `get_imagery` component service. However, the get_imagery generic function (collection of redundant methods) has as a prerequisite (sat1 = Over(region) or sat2 = Over(region)). If, for example, neither satellite is in the desired position, method reconfiguration must decide which satellite to position. Suppose sat1 is selected and its `position` method invoked. Titan continually monitors the progress of the sat1 component as it attempts to reach the goal of being over the target region. If the satellite is not making satisfactory progress, or is estimated to be in the Faulty mode, then the control sequencer invokes mode reconfiguration to attempt to restore sat1 to working order, or if this fails, invokes safe, decision theoretic method dispatch to find an alternative method sequence, as described below.

2.5 Mode Estimation

Mode estimation (ME) incrementally tracks the set of state trajectories that are consistent with the component service model, the sequence of observations and method invocations sent to these services. For example, suppose the deductive controller is trying to maintain the configuration goal "sat1 = Over(region)."

We frame ME as an instance of belief state update for a HMM. It incrementally computes the probability of state s_i at time $t + 1$ using a combination of forward prediction from the model and correction based on the observations:

$$p^{(\bullet t+1)}[s_i] = \sum_{j=1}^{n} p^{(t\bullet)}[s_j] P_{\mathrm{T}}\left(s_i \mid s_j, \mu^{(t)}\right)$$

$$p^{(t+1\bullet)}[s_i] = p^{(\bullet t+1)}[s_i] \frac{P_{\mathrm{O}}\left(o_k \mid s_i\right)}{\sum_{j=1}^{n} p^{(\bullet t+1)}[s_j] P_{\mathrm{O}}\left(o_k \mid s_j\right)}$$

where $P_{\mathrm{T}}(s_i|s_j,\mu^{(t)})$ is defined as the probability that the plant transitions from state s_j to state s_i, given control actions $\mu^{(t)}$, and $P_{\mathrm{O}}(o_k|s_i,)$ is the probability of observing o_k in state s_i. Probability $p^{(\bullet t+1)}[s_i]$ is conditioned on all observations up to $o^{(t)}$, while $p^{(\mathrm{t+1}\bullet)}[s_i]$ is also conditioned on the latest observation $o^{(\mathrm{t+1})} = o_k$.

Our approach is distinguished in that the plant HMM is encoded compactly through concurrency and constraints. The number of states is exponential in the number of components, which reaches a trillion states for even our simple example. Hence, ME enumerates the consistent trajectories and states in order of likelihood using an efficient procedure called *Conflict-directed A** [14]. This offers an any-time approach, which stops enumeration when no additional computational resources are available. Titan employs an extremely space and time efficient *compiled* mode estimation capability [12]which tracks belief states rather than trajectories, and eliminates online propositional satisfiability tests by pre-compiling the plant model using an efficient, structure-based prime implicate algorithm.

2.6 Mode Reconfiguration

Mode reconfiguration (MR) takes as input a configuration goal $g^{(t)}$, and the most likely current state $s^{(t)}$ computed by ME, and returns a series of method invocations that progress the component services towards a maximum-reward goal state that achieves $g^{(t)}$ [16,17]. MR accomplishes this using a *goal interpreter* (GI) and a *reactive planner* (RP). GI determines a target state $s^{(t)}{}_g$ that is reachable from $s^{(t)}$ and that achieves $g^{(t)}$, while maximizing reward. It accomplishes this by having Conflict-directed A* search over the reachable states in best-first order. RP generates a command sequence that moves the plant from $s^{(t)}$ to $s^{(t)}{}_g$. RP generates and executes this sequence one command at a time, using ME to confirm the effects of each command.

For example, in our image acquisition example, given a configuration goal of "sat1 = Over(region) or sat2 = Over(region)," GI selects one of the two satellites and initiates a position reorientation maneuver, which controls the satellite's thrusters.

2.7 Model-based Reactive Planning

Having identified which satellite to orient, one might imagine that achieving the configuration is a simple matter of calling a set of open-valve and close-valve routines which supply fuel to the satellite thrusters, in order to produce a desired amount of thrust. In fact, this is how Titan's predecessor, Livingstone, performed MR. However, much of the complexity of MR is involved in correctly commanding each component to its intended mode, through lengthy communication paths. For example, Figure 9 shows the communication paths to a satellite's thruster valve. The flight computer sends commands to a bus controller, which broadcasts these commands over a 1553 bus. These commands are received by a bank of device drivers, such as the propulsion drive electronics (PDE). Finally, a device driver translates the commands to analog signals that actuate its device.

The following is an example of a scenario that a robust close-valve routine should be able to handle. The corresponding procedure is automatically generated by Titan's RP:

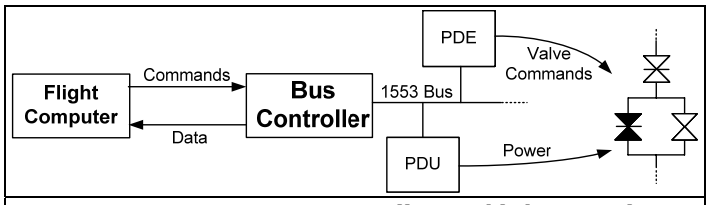


Figure 9 - A reconnaissance satellite establishes complex interaction paths from the flight computer to its thrusters.

To ensure that a valve is closed, the close-valve routine first ascertains if the valve is open or closed, by polling its sensors. If it is open, it broadcasts a "close" command. However, first it determines if the driver for the valve is on, again by polling its sensors, and if not, it sends an "on" command. Now suppose that shortly after the driver is turned on, it transitions to a resettable failure mode. The valve routine catches this failure and then before sending the "close" command, it issues a "reset" command. Once the valve has closed, the close-valve routine powers off the valve driver, to save power. However, before doing so, it changes the state of any other valve that is controlled by that driver; otherwise, the driver will need to be immediately turned on again, wasting time and power.

This example highlights several key challenges: distributed services are often controlled indirectly through other services, they can negatively interact and hence need to be coordinated, they fail and hence need to be monitored, and when they fail they must be quickly repaired. To address these challenges, Titan's RP precompiles a set of procedures that form a *goal-directed universal plan*, specifying responses for achieving all possible target states, starting in all possible current states. These procedures constitute a set of compact concurrent policies, one for

each component, and are generated by exploiting properties of causality and reversibility of action [16]. When two components are co-dependent, they must be composed into a single policy. An exponential growth in this composition, is avoided by symbolically encoding these policies using Binary Decision Diagrams [17]. This is in contrast to explicit universal plans, whose size is exponential in the number of components.
The mode reconfiguration and reactive planning capabilities just described perform system reconfiguration at the engineering level, by reasoning from component system models in order to restore function through component reconfiguration or repair. In the next section describe the control sequencer which we have extended with a system level capability for consistent method selection and dispatching that is able to detect and regenerate faulty methods.

Self Deprecation and Regeneration through Predictive Method Dispatch

In model-based programming, the execution of a method will fail if one of the service components it relies upon irreparably fails. This in turn can cause the failure of any method that relies upon it, potentially cascading to a catastrophic and irrecoverable system-wide malfunction. Titan's control sequencer enhances robustness by continuously searching for and deprecating any requisite method whose successful execution relies upon a component that is deemed faulty by mode estimation, and deemed irreparable by mode reconfiguration. For example, mode estimation may determine that Satellite 1's thruster valve is stuck closed, and mode reconfiguration determines that it is irreparable, hence the control sequencer deprecates the get_imagery method (Figure 5) that relies on Satellite 1.

Without additional action, a deprecated method will cause the deprecation of any method that relies upon it, potentially cascading to catastrophic system-level malfunction. As we demonstrated with get_imagery, model-based programmers specify redundant methods for achieving each desired function. When a requisite method is deprecated, Titan's control sequencer attempts to regenerate the lost function proactively, by selecting an applicable alternative method, while verifying overall safety of execution. This was demonstrated above for the reconnaissance example. Titan performs this in a *safe* manner, using an approach we call *predictive method dispatch.*

Titan's predictive method dispatch builds upon concepts taken from robotic execution languages, such as RAPS, [4], ESL[6] and TDL[5], which were developed in order to robustly monitor and control embedded systems with highly uncertain behavior A key claim of these languages is that they introduce fault adaptation at the system-level, by expressing and selecting, as the situation requires, between functionally redundant methods (e.g, "repair by switching to backup, or by power cycling"). These execution languages provide the programmer with greater control over the regeneration process by allowing the programmer to specify a much focused set of contingencies that the executive chooses from on the fly.

These executives, however, can be *unsafe*; they select methods on the fly *without analyzing the future consequences* of their choices. Hence it is possible for the executive to select an execution path with no successful completion, or that cannot satisfy all timing constraints.

```
1  Take_Image()[400,900] :: {
2     choose {
3        {
4           do {
5              Slew_spacecraft(SAT1)[350,950]
6           } maintaining SAT1_OK
7        } with reward 100,
8        {
9           do {
10             Slew_spacecraft(SAT2)[350,750]
11          } maintaining SAT2_OK
12       } with reward 90
13    };
14    do {
15       Acquisition(TARGET)[40,100]
17    } watching ACQUIRED;
18    do {
19       Transmit(DATA)[10,50]
20    } maintaining DOWNLINK_OK
21 }
```

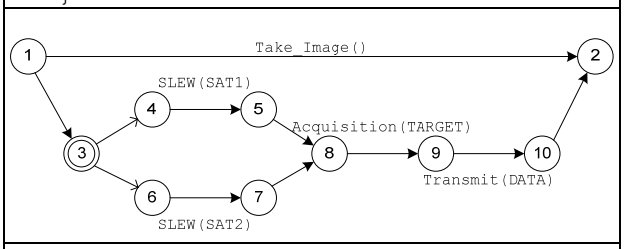


Figure 10 - An RMPL method and its TPN

Titan's control sequence provides a regenerative method dispatch capability that ensures safe execution [22, 23] through a *predictive dispatch* mechanism. Predictive method dispatch continually plans out sequences of future method invocations, and analyzes these sequences to ensure that the combined set of methods produce the intend effect. Predictive dispatch accomplishes this using forward looking search and scheduling algorithms, similar to modern, temporally flexible planners (e.g., Europa[2]). Predictive method dispatch, however, is much more efficient than traditional planning, because the space of options that it considers is limited to the possible threads of execution through the RMPL program. This speeds response and mitigates risk.

Concurrent invocations of services are often coordinated through metric timing constraints. RMPL specifies temporal constraints by bounding the time of execution of any RMPL expression, using A[l,u]. RMPL specifies multiple methods and contingent plans through a nondeterministic or decision theoretic choice expression *choose{A with reward Ra, B with reward Rb}*. Titan's treatment of reward is discussed in the next section. An example of an RMPL control program with redundant methods was given in Figure 5 for the reconnaissance scenario. A choice is made between the two methods through an implicit **choose**. An alternative RMPL program, Take_Image, with an explicit **choose** is shown in Figure 10.

Predictive method dispatch compiles these RMPL programs into a graph-based representation, called a *temporal plan network,* which compactly represents all possible threads of execution of an RMPL program, and all resource constraints and conflicts between concurrent activities. Figure 10 presents an example of a TPN corresponding to the `Take_Image` RMPL program. Predictive method dispatch "looks" using efficient network search algorithms to find maximum utility threads of execution through the TPN that are temporally consistent. The result is similar to a partially ordered temporal plan, but represents a set of safe execution threads through the RMPL program. Method dispatch then "leaps" by executing the plan using a temporally flexible dispatching algorithm [20] that adapts to uncertainties in execution time.

More specifically, *predictive method selection* first searches until it finds a set of methods that are consistent and schedulable. It then invokes the dispatcher which passes each activity to

Titan's deductive controller as configuration goals, according to a schedule consistent with the timing constraints. If the deductive controller indicates failure in the activity's execution, or the dispatcher detects that an activity's duration bound is violated, then method selection is reinvoked. The control sequencer then updates its knowledge of any new constraints and selects an alternative set of methods that safely completes the RMPL program.

Self-Optimizing Methods through Safe, Decision-Theoretic Dispatch

In addition to failure, component performance can degrade dramatically, reducing system performance to unacceptable levels. To maintain optimal performance, Titan's predictive method dispatch has been extended to decision-theoretic method dispatch, which continuously monitors performance, and selects the currently optimal available set of methods that achieve each requisite function.

2.8 Optimal, Continuous Execution under Limited Resources

To develop our approach to achieving optimal, continuous execution under limited resources, we first review a fast optimal search algorithm, Incremental A*. Next we show how optimal search algorithms are incorporated into our framework. Lastly, we discuss how Incremental A* and a memory-bounded search method, SMAG*, have been modified and incorporated to achieve optimal, continuous execution under limited resources.

Optimal Planning of TPNRM

In order for the planner to select the optimal plan for safe execution from the TPNRM, we had to address how the TPN of the TPNRM would be searched and how the roadmap of the TPNRM affects this search.

There were two main challenges to searching the TPN. First, an effective representation of the plan space needed to be defined such that an optimal algorithm, such as A*, can find the optimal plan. This plan space had to be compact for memory efficiency. Second, an effective evaluation function had to be defined for determining partial-solution cost. Through the use of heuristic estimates, the search is focused towards better solutions without sacrificing optimality. The first challenge was addressed by transforming the TPN structure of the TPNRM into a search tree similar to that of a dynamic CSP. The second challenge was addressed by taking advantage of the hierarchical structure of the TPN and preprocessing the costs of the constituent nodes in the TPN to be estimates in the search tree.

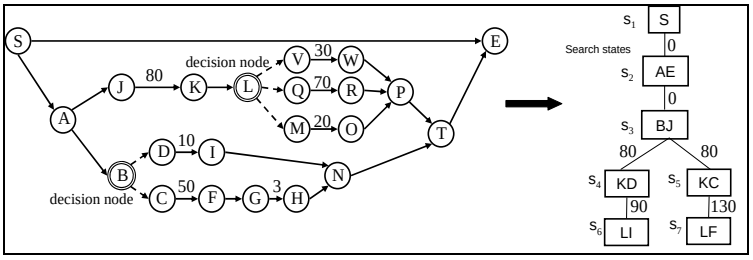


Figure 11 - Input TPN and a partial search tree.

A search state represents one or more TPN nodes at the same depth in the TPN. Initially, the start node is marked with a depth of 0. In general, the depth of node *t* is the number of nodes in the path from the start node to node *t*. Search states are created during the optimal-pre planning process and added to the search tree. In Figure 11, a TPN input (left) and corresponding search states *s1*-*s7* (right) is shown.

The structure of the search tree is determined by the decision and non-decision nodes in each search state. Each search state is created by expanding the parent search state for all permutations of TPN choice node decisions. That is, the search tree branches (creates more than one child) if the state being expanded refers to at least one decision node. For example, the search state, s_3 refers to TPN nodes *B* and *J*. When s_3 is expanded the search branches on decision node *B*, which has two choices – node *C* or node *D* – creating two child search states, s_4 and s_5.

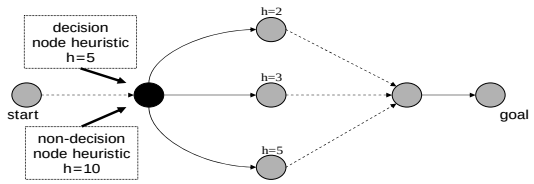


Figure 12 - Heuristic for decision and non-decision node

The cost on the arc in the search tree of each newly created child state is based on the arc costs in the parent being expanded and the sum of all costs on TPN node arcs traversed in order to create the child search state. Heuristic values are calculated differently for decision and non-decision nodes. For the expansion of non-decision nodes, heuristic values are calculated by adding the heuristic values of the fringe TPN nodes in the search state. For the expansion of decision nodes, heuristic values are calculated by taking the maximum of the heuristic value of the fringe TPN nodes in the search node. The maximum heuristic value, although potentially not a good estimate, ensures that in the estimate to the goal nodes are not double counted. An example of how these heuristics are calculated for a given non-decision and decision TPN nodes is given in Figure 12.

The roadmap in the TPNRM contributes to the optimal search by considering the path cost to a TPN activity. An optimal activity plan may not be globally optimal when paths taken into considerations there maybe an expensive or infeasible path involved in the activity plan. During the optimal search, the roadmap is queried such that paths cost can be returned to the search algorithm. When a location constraint is encountered during the optimal planning process for the TPN, the algorithm switches between the path planning space and the TPN space shown in Figure 8. This allows path planning to occur for a particular activity and provides dynamic path updates to the TPNRM. The updated costs from the roadmap provide new information about the terrain that is used to focus the activity search.

Pruning TPNRM Search Tree for Limited Memory

Autonomous agents used in mission plans are required to frequently collect and store vital sensor information, either for diagnosis, planning, or science. Therefore, not only is it necessary to optimize the speed of plan generation but also to optimize the system's memory usage. The decision making and automated reasoning of an autonomous agent's optimal planner must be able to operate within the specified memory requirements. To address the problem of memory limitations for the optimal planner, we built upon the simplified memory-bounded A* graph (SMAG*) search. The modified SMAG* algorithm optimally searches the TPN but operates within the confines of limited memory .

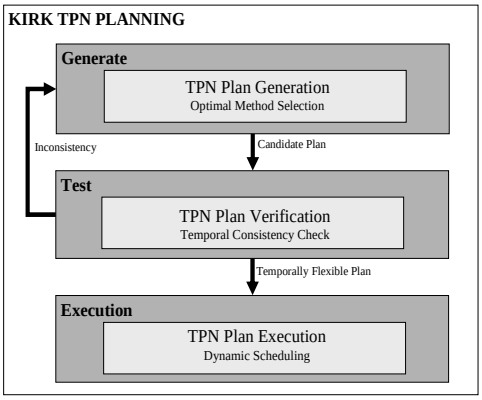


Figure 13 - Kirk Planning Diagram

SMAG* operates within memory constraints by pruning the lowest-utility states when memory limitations are reached. To retain optimality despite the removed nodes, SMAG* keeps track of two lists of nodes. One list contains nodes all of whose descendants have been explored or are in memory. The second list contains nodes whose descendants have been removed due to memory limitations. Nodes are moved between lists as their descendants are added and removed from memory. The algorithm only terminates when it reaches the optimal solution or all nodes in the search tree have been considered.

2.9 Continuous, Safe Fault Adaptation and Optimization

Continuous Optimal Planning for TPNRM

Incremental Methods for Fast Continuous Execution

For an autonomous system to achieve timely execution it must be capable of *continuous planning*: adapting quickly to a dynamic unpredictable environment. This section describes a fast planning system that supports continuous planning, temporal flexibility, and optimality. Fast planning is achieved by applying both optimal and incremental methods to the TPN planning framework. Generally, planning is a series of generate and test procedures, shown in Figure 13. A candidate plan needs to be intelligently selected by means of optimal search, and then tested for consistency (in our case dynamic schedulability). To facilitate our need for fast continuous planning, we apply incremental methods to both generate and test phases of TPN planning.

Incremental Plan Verification

A key feature of temporally flexible planning is an algorithm that can perform fast temporal reasoning. Dynamic environments require frequent temporal modifications of the candidate plan, and hence need frequent temporal consistency checks to ensure dynamic schedulability. Since successive searches through these graphs differ only slightly, often by a few nodes or arcs,

work already computed for one graph can be reused again to determine the consistency of a new graph. The ITC algorithm [16] performs a single-source shortest-path algorithm that detects

negative cycles in order to evaluate dynamic schedulability. It utilizes incremental methods borrowed from truth maintenance systems (TMS) to perform fast updates. On shortest path improvements, ITC simply updates the graph's affected successor nodes. When a shortest path gets worse, ITC uses the idea of supports from TMS to quickly determine which nodes need to be re-examined and updates those nodes accordingly. ITC demonstrates a speed improvement of approximately an order of magnitude over the leading non-incremental algorithm on cooperative scenarios with autonomous vehicles using the Kirk.

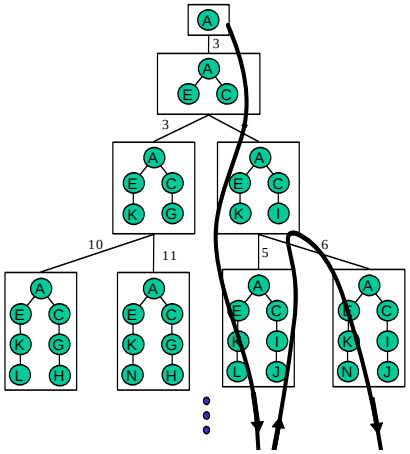


Figure 14 - Search path taken by Incremental A*

Optimal Path Planning Using Incremental A*

Most path planning search techniques are aimed to solve one-shot problems, such has what is the best path to get from point A to point B. When a problem requires additional searches, the search algorithm simply starts over from scratch. This is very inefficient because searching for the best path in a dynamic environment is often a repetitive process and there is significant overhead in re-examining unchanged nodes.

Incremental A* decreases the repeated search effort by only re-computing changed start distances. During the first search of a graph, Incremental A* behaves exactly like A* search. It finds the optimal solution by maintaining a queue of all potential paths and expands paths in the order of least cost. When an edge changes, Incremental A* only updates the affected nodes, and the optimal plan can be found using a minimal number of node expansions.

For every node, Incremental A* keeps a notion of *local consistency*, meaning whether or not the node is up to date based on its known start distance. If some particular path cost has changed, the nodes affected by the changed edge are labeled locally inconsistent. The algorithm is re-run on all locally inconsistent nodes, and when these nodes are updated with new start distances, the updates are propagated to their respective neighbors. It follows that only affected neighbors' costs need to be recomputed.

Incremental Plan Generation
In order to generate a new feasible plan due to changes in the environment, a high-level strategy must be quickly selected from the plan specification. Consequently, the TPN structure must be reexamined. Incremental A* increases the speed of these successive searches by updating only the nodes of the TPN that need to be reconsidered for finding the next plan, but also preserves optimality by utilizing A* techniques.

To perform incremental plan generation using Incremental A* for TPN planning, we transform the TPN into a dynamic CSP search tree. Based on this search tree, Incremental A* quickly updates and returns the next best TPN plan. Figure 14 shows the path that is taken by the incremental algorithm to find an optimal plan. The example search tree shown is the same search tree, but at each search state the explicit path that was taken to create the state is also shown to better illustrate the plans. Once this plan search tree is constructed, the Incremental A* algorithm traverses the tree until it reaches a plan that contains the goal. If the planning algorithm discovers an inconsistency or an environmental change, it updates the changed nodes and continues to search the current best branch. For example, suppose that the planner discovers that partial plan *A(EKL, CIJ)* is infeasible; then the cost of the arc to this partial plan is set to ∞. The Incremental A* algorithm uses this new information to backtrack (shown with the line looping back) and finds the next best optimal path to traverse, plan *A(EKN, CIJ)*. The new partial candidate plan then continues to be expanded until the new optimal plan is found. Since Incremental A* uses local updating methods, the entire plan structure tree does not need to be traversed in order to find the next best plan significantly speeding up both the plan generation and replanning process.

Integration of optimal memory-bounded techniques of SMAG* and optimal incremental search techniques over the TPN is done by running the incremental search algorithms under the memory-bounded algorithm. Thus, when new information is discovered the incremental search algorithm quickly finds the most optimal plan that is available in memory.

2.10 Comparison with Current Technology

Self-adaptive software has been successfully applied to a variety of tasks ranging from robust image interpretation to automated controller synthesis [27]. Our approach, which is described below, builds upon a successful history of hardware diagnosis and repair.

2.11 Model-based Programming of Hidden States

The reactive model-based programming language (RMPL) is similar to reactive embedded synchronous programming languages like Esterel [9]. In particular, both languages support conditional execution, concurrency, preemption and parameter less recursion. In addition, like Estrerel, RMPL is being designed as an executable specification language. This means that the program is both a specification that may be formally verified, and a program that is executable without manual translation.

The key difference is that in embedded synchronous languages, programs only read sensed variables and write to controlled variables. In contrast, RMPL specifies goals by allowing the programmer to read or write ``hidden" state variables, i.e. states that are not directly observable or controllable. It is then the responsibility of the language's model-based execution kernel to map between hidden states and the underlying system's sensors and control variables. Allowing the programmer to directly read hidden variables eliminates the need to manually code monitoring and diagnostic decision trees, whose function is to deduce the hidden health state of the system. Likewise, allowing the programmer to directly write hidden variables eliminates the need to manually code repair, reconfiguration and control codes, whose responsibility is to reliably place a system in an intended state, despite the failure of supporting components. Together this reduces dramatically the amount of self-regeneration code that must be manually coded.

2.12 Predictive and Decision-theoretic Dispatch

RMPL supports nondeterministic or decision theoretic choice, plus flexible timing constraints. Robotic execution languages, such as RAPS, [4], ESL[6] and TDL[5], offer a form of decision theoretic choice between methods and timing constraints. In RAPS, for example, each method is assigned a priority. A method is then dispatched, which satisfies a set of applicability constraints while maximizing priority. In contrast, RMPL dispatches on a cost that is associated with a dynamically changing performance measure. In RAPS timing is specified as fixed, numerical values. In contrast, RMPL specifies timing in terms of upper and lower bound on valid execution times. The set of timing constraints of an RMPL program constitutes a Simple Temporal Network (STN). Dechter and Meiri showed that, whether or not a consistent execution exists, can be determined by converting the STN to a directed graph, called a distance graph, and by solving two single source shortest path problems on the graph.

The choice operator and timing constraints can negatively interact. In particular, a set of choices may lead to an execution whose timing constraints are unsatisfiable. Execution languages like RAPS, ESL and TDL perform method selection on an as needed basic, and do not check to see if a feasible execution exists, given a particular method selection. RMPL execution is unique in that it predictively selects a set of future methods whose execution is temporally feasible.

2.13 Probabilistic Concurrent Constraint Automata

Probabilistic Concurrent Constraint Automata (PHCA) extends Hidden Markov Models (HMMs) by introducing four essential attributes. First, the HMM is factored into a set of concurrently operating automata. Second, probabilistic transitions are treated as conditionally independent. Third, each state is labeled with a logical constraint that holds whenever the automaton marks that state. This allows an efficient encoding of co-temporal processes, which interrelate states and map states to observables. Finally, a reward function is associated with each automaton, and is treated as additive. CCA are also related to Partially Observable Markov Decision Process (POMDP) in the use of probabilistic choice, reward and hidden states.

2.14 Constraint-based Trellis Diagram

Mode estimation encodes PHCA as a constraint-based trellis diagram, and searches this diagram in order to estimate the most likely system diagnosis. This encoding is similar in spirit to a SatPlan/Graphplan encoding in planning. The first differentiating factor is the behavioral complexity of a PCCA, in comparison to STRIPS plan operators. The second differentiator is the search for a set of most likely solutions, rather than satisfying solutions. In a sense, probabilistic SAT plan is a special case of PCCA in which the constraints are posed as a stochastic satisfiability problem.

2.15 Reactive Planning

Titan's reactive planner inherits from the reactive planning technique used in Burton [16]. Burton generates a form of "universal plan," which specifies a next action to perform, in order to achieve a specified goal state, given a current state. Burton is unique for its use of structural properties in order to decompose the universal plan into a compact set of hierarchical universal plans. While compact, this method is restricted to a class of systems that satisfy an acyclicity condition on the component input/output relationships. Titan expands coverage by relaxing the acyclicity condition. Universal plans for cyclic subsets of a system are encoded compactly using Reduced, Ordered Binary Decision Diagrams (ROBDDsP, similar to symbolic reactive planning, and then composed into a compact hierarchical encoding, based on Burton.

2.16 Unified representation for constraints, uncertainty and reward

Titan's mode estimation and mode reconfiguration functions are implemented by solving a restricted form of a constraint optimization problem that we call an *optimal CSP*. An optimal CSP has an objective function on a subset of the variables, called *decision variables*, and is preferentially independent. All constraints and variables are restricted to finite domains. An optimal CSP is a subset of a more general class, called *Semi-ring CSPs.* Several extensions to CSPs have been proposed that incorporate some notion of cost. One class aims at incorporating uncertainty by specifying a ``degree" to which a combination of values is allowed or not. [DFP93] propose an extension called Fuzzy CSPs, where each tuple or constraint is associated with a level of preference. [FW92] provide an extension to CSPs that allows them to model overconstrained problems. [DDP90] extends CSPs to optimization problems by specifying a weight for each tuple. [FL93] introduces probabilistic CSPs to model situations where each tuple of a constraint has a certain probability, independently of the other constraints. In [16], it is shown that these extensions can be viewed as instances of the Semiring-CSP framework forming the representational basis of our approach.

3. Accomplishments

Modeling / Programming Tools

We substantially enhanced our model-based modeling and programming tools. Specific enhancements made include: parameterized modes, parameterized methods, and inheritance of model specifications. There were numerous syntactic changes to RMPL in order to support decision-theoretic method dispatch.

Method deprecation

Detection by mode estimation that the preconditions of a method are not satisfied lead to that method being removed from consideration by the method dispatch mechanism.

Method Regeneration

Once a method is deprecated, diagnosis and repair is initiated. The basic algorithms for diagnosis and repair are based on the Titan model-based executive developed at MIT under the DARPA Mobies program.

Decision-theoretic method dispatch

Algorithms for decision-theoretic method dispatch were implemented efficiently. Dispatch now relies on method deprecation and regeneration, as well as new features to both specify and to estimate the timeliness and reliability of services and methods under consideration.

MIT Testbed

All work was performed at the MIT Computer Science and Artificial Intelligence Laboratory (CSAIL).

3.1 Products of the Research

The products of this work are in the form of representation languages, software and know-how. The representations and software are embodied within the *model-based programming toolkit*. The toolkit includes an implementation of the Reactive Model-based Programming Language (RMPL), the Titan *mode-based executive,* and the temporal planner (KIRK).

RMPL is compiled to an efficient internal representation of a subsystem's control methods and probabilistic models of the component services it relies upon. At runtime, the Titan *model-based executive* uses its leading edge *mode estimation* algorithm to dynamically assess the health and performance of the component services. It then uses *safe dynamic method dispatch* to select optimal combinations of control methods for achieving intend function, given the results of mode estimation. Method execution results in a sequence of *configuration goals*, which specify

desired functions to be provided by the underlying component services. Finally, Titan's *mode reconfiguration* capability generates commands to reconfigure or repair the component services in order to satisfy the specified configuration goals. Throughout this process the executive uses mode estimation to analyze its progress with respect to a current set of goals and uses both mode reconfiguration and safe dynamic dispatch to actively replan based on its status.

3.2 A Simple Example

To describe the machinery of Predictive Method Selection and Decision-Theoretic Dispatch in more detail, we present a simple example. In this experiment a rover is tasked with performing science operations on the surface of Mars. Specifically, in this example, the rover is asked to visit two Martian rocks and sample their composition. The primary goal is to scratch each rock using a grinder (a.k.a. rock abrasion tool) and then analyze the rock's composition. As a secondary goal, if for some reason the primary goal is unattainable, the rover should take a high resolution image of the rock.

In Figure 15, below, we show how this simple example can be encoded as a Temporal Plan Network (TPN). A TPN [8] can be viewed as a generalization of a Simple Temporal Network (STN) that supports multiple redundant methods, method deprecation and regeneration, and optimal planning.

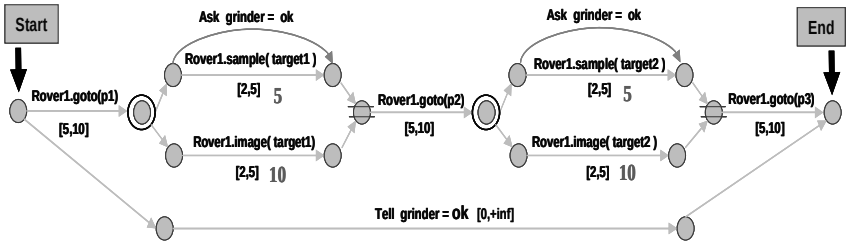
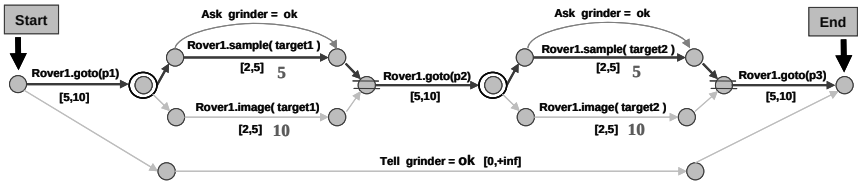


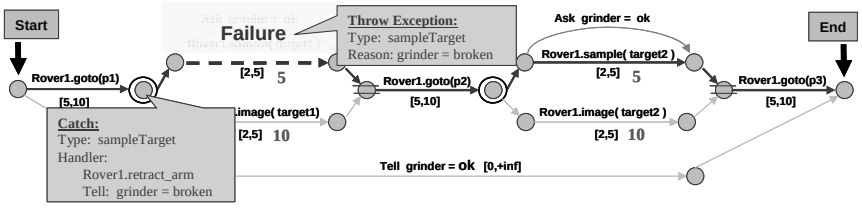
Figure 15 - Temporal Plan Network encoding of the MER Rover Example.

In a TPN, nodes represent instantaneous time events, and arcs represent activities with duration. The duration of an activity is constrained by a lower and upper timebound, specified in brackets, [lb,ub]. By default, a blank arc has [0,0] timebounds. Redundant methods in a TPN are denoted by a double circle and a circle with two horizontal lines. One thread of execution must be chosen between each pair of double circle node and double line node in the TPN. This construct allows the autonomous system to select from multiple redundant methods at runtime. Method deprecation and regeneration is enabled through Ask and Tell operators. For example, in Figure 15, we enforce that the grinder be in working condition if we wish to sample the composition of a rock. This condition is enforced by placing an arc over each sample(Target) activity which asks that the condition grinder = ok holds true during the entire execution of the activity. The dispatcher enforces that an Ask condition is satisfied by covering it with a Tell condition. For example, at the bottom of the plan in Figure 15, we tell the plan that grinder = ok for the duration of the plan. Additionally, a cost

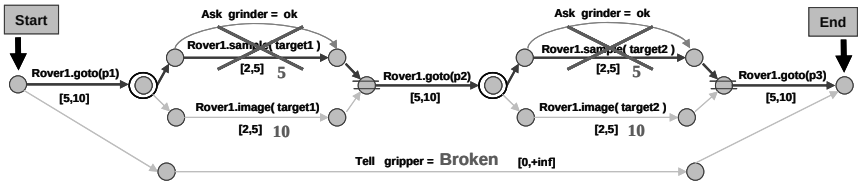
Step 1: Predictive Method Selection:
- initially select a set of methods that are consistent, schedulable, and optimal



Step 2: Deductive Controller indicates failure of an activity during execution
- the grinder fails and can't sample target 1



Step 3: Method Deprication:
- update the plan with new information (grinder = broken), sample(Targets) is now depricated



Step 4: Reinvoke Predictive Method Dispatch:
- choose the optimal set of methods, while avoiding any depricated methods

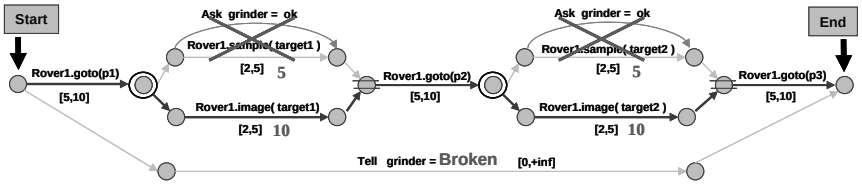


Figure 16 - A walkthrough example.

can be associated with each activity in the TPN. This allows the planner to choose the set of available methods that minimizes cost, or analogously, maximizes reward. The cost of an activity is denoted by a number, and as shown in Figure 16 we give each sample(Target) activity a cost of 5 and each image(Target) activity a cost of 10. By giving each sample(Target) activity a lower cost than the associated image(Target) activity, we are abiding by the plan requirement that sampling each rock is the primary goal, and imaging each rock is just the secondary goal.

Next, we walk through each step of the MER rover example depicted above in Figure 16. Initially, in step 1, Predictive Method Selection is invoked and an optimal consistent and schedulable plan is selected. We see that among the redundant methods of sampling and imaging each rock, the planner correctly chooses to sample each rock (since this is the option with the least cost). Now that a plan is selected, it is sent to the dispatcher. In step 2, we assume that the grinder breaks, causing a failure exception to be thrown. When the failure occurs, the plan is updated with any new information (grinder = broken) and then method selection is re-invoked. However, as shown in step 3, we see that since the grinder is now broken, the Ask condition associated with each sample(Target) activity is no longer satisfied, thus the activities

must be deprecated. In step 4, predictive method selection is re-invoked, and the optimal consistent and schedulable plan is selected. As is shown, with a broken grinder, the only consistent alternative for the MER rover is to take high-resolution images of the two targets.

3.3 Results

Initial testing of the described system has been performed by augmenting the MIT MERS rover test bed. The rover test bed consists of a fleet of ATRV robots within a simulated Martian terrain. By way of example we describe one mission whose robustness has been enhanced by the system.

Two rovers must cooperatively search for science targets in the simulated Martian terrain. This is done by having the rovers go to the selected vantage points looking for science targets using the rover's stereo cameras. The rovers divide up the space so that they can minimize the time taken in mapping the available science targets in the area. The paths of the rovers are planned in advance given existing terrain maps. The plan runs without fail. Between them the rovers successfully find all of the science targets that we have placed for them to find. The scenario is shown below in Figure 17.

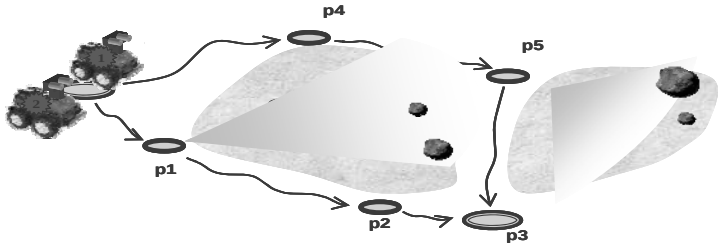


Figure 17 - Rover test bed experimental platform

In the test scenario two faults are introduced by placing a large rock that blocks rover #1's view of one of the designated areas (Figure 18). When rover #1 gets into its initial position to look for science targets its stereo cameras detect the unexpected rock obscuring its view. This results in an exception that disqualifies the current software component from looking for targets. Since the failure is external to the rover software the plan itself is invalidated. The exception is resolved by replanning which allows the both rovers to modify their plans so that the second rover observes the obscured site from a different vantage point. The rovers continue with the new plan but when rover #2 attempts to scan the area for science targets the selected vision algorithm fails due to the deep shadow being cast by the large rock. Again an exception is generated but in this case a redundant method is found – a vision algorithm that works well in low light conditions. With this algorithm the rover successfully scans the site for science targets. Both rovers continue to execute their plan without further failure.

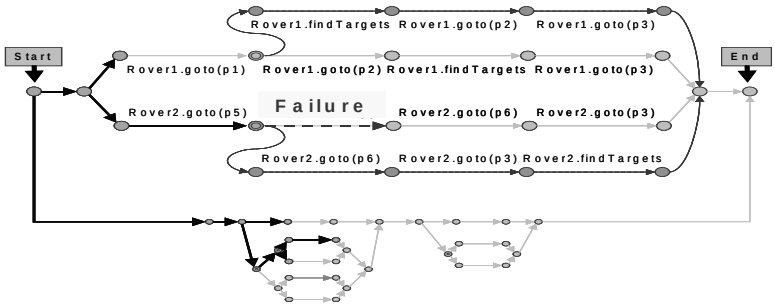


Figure 18 - The TPN for the two-rover exploration plan. Failure due to an obscuration (rock) results in automatic online replanning so that the mission can continue.

4. Conclusions

We have extended a system capable of diagnosing and reconfiguring redundant hardware systems so that instrumented software systems can likewise be made robust. Modeling software components and their interconnections poses a high modeling burden. Results of our early experiments are encouraging but much work remains to extend the current experimental system to cover the full range of software practice.

1. **Pervasive system robustness by composing concurrent fault aware processes.**
   In open systems, failures can occur within any subsystem, process or component of the system, not just at its perimeter. To achieve robustness for open systems, we enable every process to be fault aware, by recognizing and adapting to failure. In contrast to traditional, centralized approaches, our approach supports fault-aware processes that operate concurrently while communicating across a network, and that operate through a layered architecture within a single process.
2. **Fault-adaptive processes through model-based program execution.**
   To achieve robustness pervasively, fault adaptive processes must be created with minimal programming overhead. *Model-based programming* elevates this task to the specification of the intended state evolutions of each process. A *model-based executive* automatically synthesizes fault adaptive processes for achieving these state evolutions by reasoning from models of correct and faulty behavior of supporting service components. This synthesis includes methods for transitioning to intended states, monitoring progress, diagnosing failure, and repairing or reconfiguring underlying components. Furthermore, the model-based executive can construct *novel recovery actions* in the face of *novel faults.*
3. **Self-deprecating methods through prognostic mode estimation.**
   As with traditional languages, model-based programs are specified in terms of a set of methods and method invocations. Execution of these methods will fail if the service components they rely upon irreparably fail. Model-based execution enhances robustness by continuously deprecating any method that is involved in the current execution plan and whose successful execution relies upon an irreparable component.
4. **Self-regenerating methods through redundant method dispatch**.
   When a method is deprecated, the model-based executive attempts to regenerate the lost

function that caused the method deprecation by reasoning about its service component models in order to repair or reconfigure the faulty services. To handle the event of permanent method deprecation, a model-based program includes a specification of redundant methods for achieving each desired function. If a deprecated method cannot be repaired, the desired functionality is regenerated dynamically by choosing a suitable redundant method and verifying correct function.

5. **Self-optimizing methods through decision-theoretic dispatch**.
   In addition to failure, component performance can degrade dramatically, reducing system performance to unacceptable levels. To maintain optimal performance, decision-theoretic method dispatch continuously monitors performance, and selects the currently optimal available method that achieves the desired function.
6. **Safe fault adaptation through method dispatch as continuous planning**
   Failures can occur at any moment and can have a disastrous effect if not immediately caught. Our approach continuously monitors failure and performance degradation, and reactively invokes its regeneration processes as required.
7. **Incorporation of fault adaptation incrementally.**
   Improving robustness of large, legacy systems must be a gradual process, whose cost can be amortized over time. *Our approach allows us to add robustness to individual software components, thus incrementally increasing the robustness of the overall system.*

4.1 Overview of Accomplishments

Research Accomplishments: We have provided software tools to add intelligent fault awareness and recovery to complex critical software systems. We have developed the novel ideas of *method deprecation* and *method regeneration* in tandem with an intelligent runtime *model-based executive* that performs automated fault management from engineering models, and that utilizes *decision-theoretic method dispatch.* Once a system has been enhanced by abstract models of the nominal and faulty behavior of its components, the model-based executive monitors the state of the individual components according to the models. If faults in a system render some methods (procedures for accomplishing individual goals) inapplicable, method deprecation removes the methods from consideration by the decision-theoretic dispatch. Method regeneration involves repairing or reconfiguring the underlying services that are causing some method to be inapplicable. This regeneration is achieved by reasoning about the consequences of actions using the component models, and by exploiting functional redundancies in the specified methods. In addition, decision-theoretic dispatch continually monitors method performance and dynamically selects the applicable method that accomplishes the intended goals with maximum safety, timeliness, and accuracy.

Beyond simply modeling existing software and hardware components, we allow the specification of high-level *methods*. A method defines the intended state evolution of a system in terms of goals and fundamental control constructs, such as iteration, parallelism, and conditionals. Over time, the more that a system's behavior is specified in terms of model-based methods, the more that the system is able to take full advantage of the benefits of model-based programming and the runtime model-based executive. Implementing functionality in terms of methods enables

method prognosis, which involves proactive method deprecation and regeneration, by looking ahead in time through a temporal plan for future method invocations.

Our approach has the benefit that every additional modeling task performed on an existing system makes the system more robust, resulting in substantial improvements over time. As many faults and intrusions have negative impact on system performance, our approach also improves performance of systems under stress. In addition, decision theoretic dispatch takes into account performance profiles of available methods, as well as the temporal requirements of the goals at hand, in order to select the best performing methods that guarantee overall correctness.

Impact: Our research has provided a well-grounded technology for incrementally increasing the robustness of complex, concurrent, critical applications. When applied pervasively, model-based execution can dramatically increase the security and reliability of these systems, as well as improve overall performance, especially when the system is under stress.

Main goals of the proposed research: We have provided the ability to incrementally and pervasively increase the robustness of complex systems. We have provided model-based programming tools that enable the high-level specification of *self-deprecating* and *self-reconfiguring* methods. We have developed model-based executives that provide *safe, optimal dispatching* of functionally redundant methods, and that reason from component service models to continuously monitor, diagnose, regenerate and optimize function.

Tangible benefits to end users: Systems embedded with executives demonstrate dramatic increases in robustness through dynamic adaptation to faults and attacks, and safe selection of alternative strategies that maintain system function. Performance is improved as subsystems are compromised or overwhelmed by using *safe optimal dispatch* to compute and select the best performing, verifiably correct implementation.

Main elements of the proposed approach: We model system components and behavior in an intuitive but formally defined language. Component *methods* specify intended state evolution over time. *Method deprecation* dynamically detects and removes methods that are no longer able to safely achieve their intent. *Method regeneration* uses model-based diagnosis and repair techniques to dynamically restore a method to health. *Decision-theoretic method dispatch* exploits functional redundancy in the system, by dynamically selecting alternative methods that work around faulty or suboptimal components.

Rationale: For two decades we have pioneered model-based technologies for execution, monitoring, diagnosis and repair, applying them successfully to domains ranging from copiers and automobiles to spacecraft. Our Livingstone model-based monitoring and repair system on NASA's Deep Space One probe marked a paradigm shift in creating self-repairing explorers, while handling dynamic uncertainty and large repair spaces[3]. Our DARPA MOBIES research addressed improved performance by aggressive compilation of models and model-based programs.

Nature of results: We developed software tools that enable model-based fault-adaptivity to be incrementally added to existing software systems.

Value of our work: Both our military and our homeland security operations rely on acquiring, transmitting, analyzing, and presenting vast amounts of data in a safe and timely manner. These critical systems are in constant danger of failure, due to both innocuous and malicious threats. Without pervasive model-based fault containment, small faults or intrusions will continue to threaten entire systems upon which our security depends. The capabilities that we developed under this program, which we demonstrated on our rover testbed platform, offer a mature approach to providing protection to critical software systems.

4.2 Further Work

Our results within this program have focused on providing robust execution of software systems within a hostile and often only marginally predictable environment. No attempt was made to model explicitly hostile attacks on software systems. Given our success with achieving robustness in the face of passively hostile environments a natural next step is to extend our model-based approach to modeling explicit hostile attacks on software systems.

5. References

Work performed under this contract was described in three publications listed below:

Publications

[1] P. Robertson and B. C. Williams, "*A Model-Based System Supporting Automatic Self-Regeneration of Critical Software*", SelfMan2005

[2] P. Robertson, R. T. Effinger, and B. C. Williams, "*Autonomous Robust Execution of Complex Robotic Missions*", IAS-9

[3] P. Robertson, B. C. Williams, "*Automatic Recovery from Software Failure*", CACM March 2006 pp41-47

Bibliography

[1] J. Casani *et al.* "Report on the Loss of the Mars Polar Lander and the Deep Space 2 Probes," Caltech JPL Techreport D-18709, March, 2000.

[2] N. Muscettola, P. Nayak, B. Pell, and B. Williams, Remote Agent: to boldly go where no AI has gone before. AIJ, 103, 98.

[3] D. Bernard, G. Dorais, E. Gamble, B. Kanefsky, J. Kurien, G. Man, W. Millar, N. Muscettola, P. Nayak, K. Rajan, N. Rouquette, B. Smith, W. Taylor, Y. Tung, Spacecraft Autonomy Flight Experience: The DS1 Remote Agent Experiment, Proceedings of the AIAA Space Technology Conference & Exposition, Albuquerque, NM, Sept. 28-30, 1999. AIAA-99-4512.

[4] R. Firby, "The RAP language manual," Working Note AAP-6. University of Chicago, 1995.

[5] R. Simmons, "Structured Control for Autonomous Robots," *IEEE Transactions on Robotics and Automation*, 10(1), 94.

[6] E Gat, "ESL: A Language for Supporting Robust Plan Execution in Embedded Autonomous Agents," In Proceedings of the AAAI Fall Symposium on Plan Execution, 1996.

[7] G. Berry, "Real-time Programming: General Purpose or Special Purpose Languages," In G. Ritter (ed.) Information Processing, 89, 1989, pp. 11-17.

[8] D. Harel, "Statecharts: A Visual Approach to Complex Systems," Sci. of Computer Programming, 8, 1987, pp. 231-274.

[9] G. Berry and G. Gonthier, "The Esterel Programming Language: Design, Semantics and Implementation," Science of Computer Programming, 18(2), 1992, pp. 87-152.

[10] V. Gupta, R. Jagadeesan, V. Saraswat. Models for Concurrent Constraint Programming. *Proceedings of CONCUR'96: Concurrency Theory*, edited by Ugo Montanari and Vladimiro Sassone, LNCS 1119, Springer Verlag, 1996.

[11] B. Williams, M. Ingham, S. Chung and P. Elliott, "Model-based Programming of Intelligent Embedded Systems and Robotic Explorers," IEEE Proceedings Special Issue on Embedded Software.

[12] Chung, S., J. Van Eepoel and B.C. Williams, “Improving Model-based Mode Estimation through Offline Compilation,” Int. Symp. on Artificial Intelligence, Robotics and Automation in Space, St-Hubert, Canada, June 2001.

[13] B. Williams and P. Nayak. A Model-based Approach to Reactive Self-Configuring Systems. In Proceedings of AAAI-98, 971-978, 1996.

[14] Williams, B.C. and R. Ragno, “Conflict-directed A* and its Role in Model-based Embedded Systems," *to appear in* Special Issue on Theory and Applications of Satisfiability Testing, Journal of Discrete Applied Math.

[15] L. Fesq *et al.,* “Model-based Autonomy for the Next Generation of Autonomous Spacecraft,” in Proceedings of the 53rd International Astronautical Congress of the International Astronautical Federation, (IAC-02), October, 2002.

[16] B. Williams and P. Nayak. A Reactive Planner for a Model-based Execution. In Proceedings 15th International Joint Conference AI, Nagoya, Japan, August 1997. IJCAI-97.

[17] Chung, S. and B. C. Williams, A Factored Symbolic Approach to Reactive Planning, 14th International Workshop on Principles of Diagnosis (DX '03), Washington D.C., June, 2003.

[18] Ingham, M. and B.C. Williams, Timed Model-based Programming: Writing Executable Specifications for Robust Executable Systems, Intl. Conference on Self Adaptive Software, 2003.

[19] Ingham, M. Robust Spacecraft Execution through Timed Model-based Programming, PhD Thesis, MIT.

[20] N. Muscettola, P. Morris, B. Pell, B. Smith. Issues in Temporal Reasoning for Autonomous Control Systems. *Autonomous Agents*. Minneapolis, MN, 1998.

[21] G. Rabideau, R. Knight, S. Chien, A. Fukunaga, A. Govindjee. Iterative Repair Planning for Spacecraft Operations in the ASPEN System. *Intl Symp on AI Robotics and Automation in Space*, Noordwijk, The Netherlands, June 1999.

[22] P. Kim, B. Williams and M. Abramson, “Executing Reactive, Model-based Programs through Graph-based Temporal Planning,” International Joint Conference on Artificial Intelligence, Seattle, WA, August 2001, pp. 487-493.

[23] Brian C. Williams et al. Model-based Reactive Programming of Cooperative Vehicles for Mars Exploration, Proceedings of the International Symposium on Artificial Intelligence and Robotics in Space, Montreal, CA, 2001.

[24] Hofbaur, M. W. and B.C. Williams, “Mode Estimation of Probabilistic Hybrid Systems,” International Conference on Hybrid Systems: Computation and Control, March 2002.

[25] Hofbaur, M. and B.C. Williams, Hybrid Estimation of Complex Systems, to appear IEEE System, Man, and Cybernetics Society Special issue of IEEE SMC Transactions - Part B Diagnosis in Complex Systems: Bridging the methodologies of the FDI and DX Communities, 2003.

[26] Funiak, S. and B. C. Williams, Multi-modal Particle Filtering for Hybrid Systems with Autonomous Mode Transitions, Proceedings SafeProcess-03, June, 2003.

[27] Robert Laddaga, Paul Robertson, Howard E. Shrobe. Introduction to Self-adaptive Software: Applications. Springer Verlag LNCS 261